JASPERREPORTS - QUICK GUIDE

JASPERREPORTS GETTING STARTED

Jasper Report is an open source java reporting engine, which unlike other reporting tools, for example, Crystal Reports, is Java based and doesn't have its own expression syntax. JasperReports has the ability to deliver rich content onto the screen, to the printer, or into PDF, HTML, XLS, RTF, ODT, CSV, TXT and XML files. As it is not a standalone tool, it cannot be installed on its own. Instead, it is embedded into Java applications by including its library in the application's CLASSPATH. JasperReports is a Java class library, and is not meant for end users, but rather is targeted towards Java developers who need to add reporting capabilities to their applications.

FEATURES OF JASPERREPORTS

Some of the main JasperReport features include:

- Has flexible report layout.

- It can present data textually or graphically.

- Developers can supply data in multiple ways.

- It can accept data from multiple datasources.

- It can generate watermarks.(A watermark is like a secondary image that is laid over the primary image)

- It can generate subreports.

- It is capable of exporting reports to a variety of formats.

JASPERREPORTS ENVIRONMENT SETUP

Downlaod the latest JAR alongwith the required and optional libraries (.ZIP file) from the site: JasperReport Library Link. .

The ZIP file includes the JasperReports JAR file alongwith the JasperReports source code, depenedent JARs and a lot of examples demonstrating JasperReport's functionality.

To start creating the reports we need to set up the environment ready. Extract the downloaded JasperReport .ZIP file to any location(In our case we have extracted it to C:\tools\jasperreports-5.0.1). The directory structure of the extracted file is as in screen below:

Setting classpath

To use the JasperReport, we need to set the following files to our CLASSPATH:

- jasperreports-x.x.x.jar, where x.x.x is the JasperReports version. This found under directory C:\tools\jasperreports-x.x.x\dist).

- All the JAR files under the *lib* subdirectory (C:\tools\jasperreports-x.x.x\lib).

At the time of installation, we used JasperReport version 5.0.1. Right-click on 'My Computer' and select 'Properties', click on the 'Environment variables' button under the 'Advanced' tab. Now update the 'Path' variable with this **C:\tools\jasperreports-5.0.1\dist\jasperreports-5.0.1.jar;C:\tools\jasperreports-5.0.1\lib;**. Now you are ready to create your reports.

> *In all the examples in this tutorial we have used ANT tasks to generate reports. The build file takes care of including all the required JARs for generating reports. Hence, setting CLASSPATH as mentioned above will only help those who wish to generate reports without using ANT.*

Jasper Managers classes

There are number of classes which will be used to compile a JRXML report design, to fill a report, to print a report, to export to PDF, HTML and XML files, view the generated reports and report design.
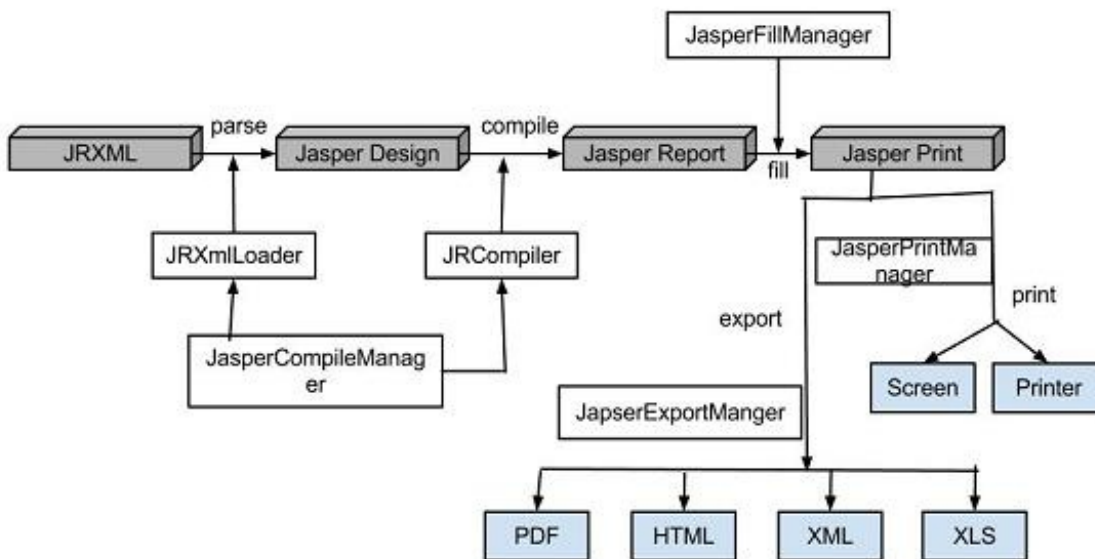
The list of these classes is:

- *net.sf.jasperreports.engine.JasperCompileManager:* Used to copile a JRXML report template.

- *net.sf.jasperreports.engine.JasperFillManager:* Used to fill a report with data from a datasource

- *net.sf.jasperreports.engine.JasperPrintManager:* Used to print the documents generated by the JasperReports library

- *net.sf.jasperreports.engine.JasperExportManager:* Used to obtain PDF, HTML, or XML content for the documents produced by the report-filling process

- *net.sf.jasperreports.view.JasperViewer:* It represents a simple Java Swing application that can load and display reports.

- *net.sf.jasperreports.view.JasperDesignViewer:* Used at design time to preview the report templates.

JASPERREPORTS LIFE CYCLE

The main purpose of JasperReports is to create page oriented, ready to print documents in a simple and flexible manner. The following flow chart depicts a typical work flow while creating reports.
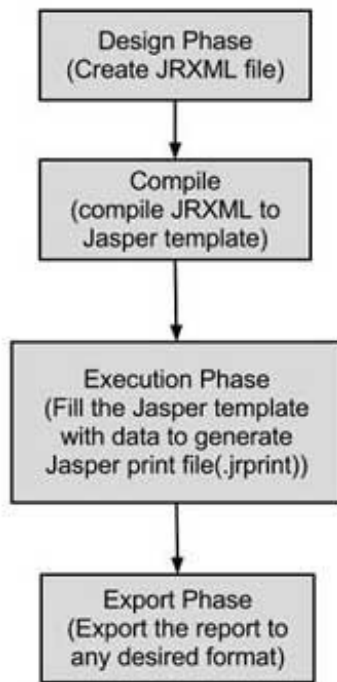
As in the image the life cycle has following distinct phases

1. Designing the report In this step we create the JRXML file, which is an XML document that contains the definition of the report layout. We can use any text editor or iReportDesigner to manually create it. If iReportDesigner is used the layout is designed in a visual way, hence real structure of the JRXML can be ignored.

2. Compiling the report In this step JRXML is compiled in a binary object called a Jasper file(*.jasper). This compilation is done for performance reasons. Jasper files are what you need to ship with your application in order to run the reports.

3. Executing the report(Filling data into the report) In this step data from the application is filled in the compiled report. The class net.sf.jasperreports.engine.JasperFillManager provides necessary functions to fill the data in the reports. A Jasper print file (*.jrprint) is created, which can be used to either print or export the report.

4. Exporting the report to desired format In this step we can export the Jasper print file created in the previous step to any format using JasperExportManager. As Jasper provides various forms of exports, hence with the same input we can create multiple representations of the data.

JASPER REPORT - REPORT DESIGN

The JRXML templates (or JRXML files) in JasperReport are standard XML files, having an extension of .jrxml. All the JRXML files contain tag <jasperReport>, as root element. This in turn contains many sub-elements (all of these are optional). JasperReport framework can handle different kinds of data sources.

We can create a JRXML using a text editor. The created JRXML can be previewed using the utility net.sf.jasperreports.view.JasperDesignViewer available in JasperReports JAR. This utility is a standalone Java application, hence can be executed using ANT. Here the preview shows only report expressions for obtaining the data are displayed, as JasperDesignViewer doesn't have access to the actual datasource or report parameters.

JASPER REPORT - COMPILING REPORT DESIGN

The report temlate or JRXML cannot be used directly to generate reports. It has to be compiled to JasperReport' native binary format, called Jasper file. On compiling we transform JasperDesign object into JasperReport object:
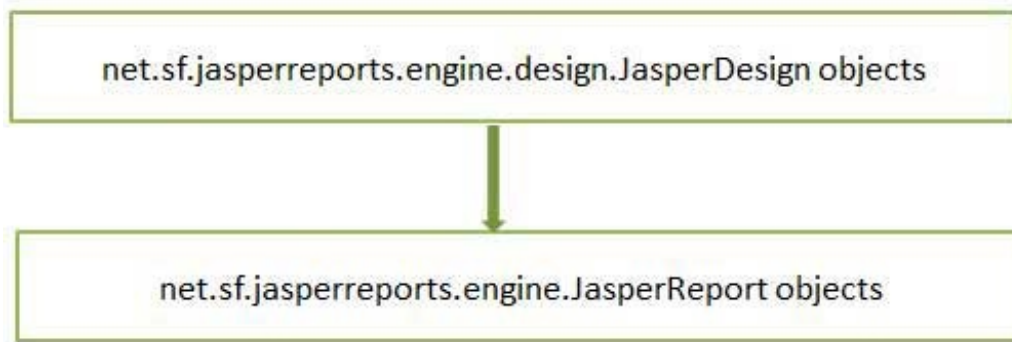
JASPER REPORT - FILLING REPORTS

Report filling process helps reporting tool to produce high quality documents by manipulating sets of data. The main input required for report-filling process is:

- **Report Template:** This is actual JasperReport file

- **Report Parameters** These are basically named values that are passed at the report filling time to the engine.

- **Data Source** We can fill a Jasper file from a range of datasources like an SQL query, an XML file, a csv file, an HQL (Hibernate Query Language) query, a collection of Java Beans, etc.

The output generated of this process **.jrprint** is a document ready to be viewed, printed or exported to other formats. The facade class *net.sf.jasperreports.engine.JasperFillManager* is usually used for filling a report template with data. This class has various *fillReportXXX()* methods that fill report templates (templates could be located on disk, picked from input streams, or are supplied directly as in-memory).

There are two categories of fillReportXXX() methods in this facade class:

1. The first type, receive a java.sql.Connection object as the third parameter. Most of the times reports are filled with data from a relational database. This is achieved by:

   - Connect to the database through JDBC.

   - Include an SQL query inside the report template.

   - JasperReports engine uses the connection passed in and executes the SQL query.

   - A report data source is thus produced for filling the report.

2. The second type, receive a net.sf.jasperreports.engine.JRDataSource object, when data to be filled is available in other forms.

JASPER REPORT - VIEW AND PRINT REPORTS

The output of the report filling process *JasperPrint objects* can be viewed using a built-in viewer component, or printed or exported to more popular document formats like PDF, HTML, RTF, XLS, ODT, CSV, or XML.

Viewing Reports

JasperReport provides a built-in viewer for viewing the generated reports in its original format. It is a swing based component and other Java applications can integrate this component without having to export the documents to other

formats in order to be viewed or printed. The *net.sf.jasperreports.view.JRViewer* class represents this visual component. This class can also be customized as per the application needs, by subclassing it.

JasperReports also has a Swing application that uses the visual component for viewing the reports. This application helps view reports in the same format as *.jrprint is produced. This Swing application is implemented in the class *net.sf.jasperreports.view.JasperViewer*. To use this, we can wrap this into an ANT target, to view the report.

Printing Reports

We can print the documents generated by the JasperReports library (in their proprietary format i.e *JasperPrint* objects) using the *net.sf.jasperreports.engine.JasperPrintManager* class. This is a facade class that relies on the Java 2 Printing API. We can also print the documents once the JasperReport documents are exported to other formats such as HTML or PDF.

JASPER REPORT - EXPORTING REPORTS

Generated reports can be exported to other formats like PDF,HTML and XLS. Facade class *net.sf.jasperreports.engine.JasperExportManager* is provided to achieve this functionality. Exporting means transforming the *JasperPrint* object (.jrprint file) into different format.

REPORT PARAMETERS

Parameters are the object references that are passed during report-filling operations to the report engine. Parameters are useful for passing useful data to report engine, the data which cannot be passed through the datasource. Data like author name, title of the report etc, can be passed through parameters. A Jasper report template or JRXML template can have zero or more parameter elements.

Parameter Declaration

Parameter declaration is simple as follows:

```
<parameter name="exampleParameter"  />
```

The name attribute

The *name* attribute of the <parameter> element is mandatory. It references the parameter in report expressions by name. Parameter name should be a single word. It should not contain any special characters like dot or comma.

The class attribute

The *class* attribute is also mandatory and it specifies the class name for the parameter values. Its default value is *java.lang.String*. This can be changed to any class available at runtime. Irrespective of the type of a report parameter, the engine takes care of casting in the report expressions in which the $P{} token is used, hence making manual casts unnecessary.

The report parameter values are always packed in a java.util.Map object, which has the parameter name as its key. Report parameters can be used in the query string of the report, so as to further customize the data set retrieved from the database. These act like dynamic filters in the query that supplies data for the report.

Built-in Parameters

Following are the pre-defined report parameters, ready to use in expressions:

Parameter Name	Description
REPORT_PARAMETERS_MAP	Contains a map with all user defined and built-in parameters
REPORT_CONNECTION	This points to user supplied java.sql.Connection used for JDBC datasources
REPORT_DATA_SOURCE	This is a user supplied instance of JRDataSource representing either one of the built-in data source types or a user-defined one
REPORT_MAX_COUNT	This is a *java.lang.Integer* value, allowing the users to limit the records from datasource.
REPORT_SCRIPTLET	This points to *net.sf.jasperreports.engine.JRAbstractScriptlet* and contains an instance of the report scriptlet provided by the user
REPORT_LOCALE	This a *java.util.Locale* instance, containing the resource bundle desired locale
REPORT_RESOURCE_BUNDLE	This points to *java.util.ResourceBundle* object and contains localized messages
REPORT_TIME_ZONE	This a *java.util.TimeZone* instance, used for date formatting
REPORT_VIRTUALIZER	This is an instance of *net.sf.jasperreports.engine.JRVirtualizer* object, and used for page virtualization (optimize memory consumption)
REPORT_CLASS_LOADER	This is a *java.lang.ClassLoader* instance to be used during the report filling process to load resources such as images, fonts and subreport templates
IS_IGNORE_PAGINATION	If set to *java.lang.Boolean.TRUE* the report will be generated on one long page and page break will not occur

DATASOURCES

Datasources are structured data container. While generating the report, Jasper report engine obtains data from datasources. Data can be obtained from databases, XML files, arrays of objects and collection of objects. The fillReportXXX () method expects to receive a data source of the report that it has to fill, in the form of **net.sf.jasperreports.engine.JRDataSource** object or a **java.sql.Connection** (when the report data is found in a relational database).

The JRDataSource interface has only two methods, which should be implemented:

1. *public boolean next() throws JRException;*

    At the report filling time, this method is called on the data source object by the reporting engine when iterating through the data.

2. *public Object getFieldValue(JRField jrField) throws JRException;*

    This method provides the value for each report field in the current data source record.

The only way to retrieve data from the data source is by using the report fields. There are several default implementations of the JRDataSource interface, depending on the way the records in the data source are acquired.

Datasource Implementations

Datasource	Implementation Class
JDBC	net.sf.jasperreports.engine.JRResultSetDataSource
JavaBean	net.sf.jasperreports.engine.data.JRBeanCollectionDataSource, net.sf.jasperreports.engine.data.JRBeanArrayDataSource
Map-based	net.sf.jasperreports.engine.data.JRMapArrayDataSource, net.sf.jasperreports.engine.data.JRMapCollectionDataSource
TableModel	net.sf.jasperreports.engine.data.JRTableModelDataSource
XML	net.sf.jasperreports.engine.data.JRXmlDataSource
CSV	net.sf.jasperreports.engine.data.JRCsvDataSource
XLS	net.sf.jasperreports.engine.data.JRXlsDataSource
Empty	net.sf.jasperreports.engine.JREmptyDataSource

Rewindable Data Sources

The **net.sf.jasperreports.engine.JRRewindableDataSource** extends the basic *JRDataSource* interface. It adds only one method, called moveFirst (), to the interface. This method is intended to move the cursor to the first element in the datasource.

Rewindable data sources are useful when working with subreports placed inside a band that is not allowed to split due to the isSplitAllowed="false" setting and there is not enough space on the current page for the subreport to be rendered.

All the above data source implementations are rewindable except for the **JRResultSetDataSource**, as it does not support moving the record pointer back. This poses a problem only if this data source is used to manually wrap a java.sql.ResultSet before passing it to the subreport. There is no problem, if the SQL query resides in the subreport template, as the engine will execute it again when restarting the subreport on the next page.

Data Source Providers

The JasperReports library has an interface **net.sf.jasperreports.engine.JRDataSourceProvider**. This helps in creating and disposing of data source objects. When creating a report template using GUI tools, a special tool for customizing the report's data source is needed. JRDataSourceProvider is the standard way to plug custom data sources into a design tool. A custom implementation of this interface should implement the following methods that allow creating and disposing of data source objects and also methods for listing the available report fields inside the data source if possible:

```
public boolean supportsGetFieldsOperation();

public JRField[] getFields(JasperReport report)
    throws JRException, UnsupportedOperationException;

public JRDataSource create(JasperReport report) throws JRException;

public void dispose(JRDataSource dataSource) throws JRException;
```

REPORT FIELDS

Report fields are elements which represent mapping of data between datasource and report template. Fields can be combined in the report expressions to obtain the desired output. A report template can contain zero or more <field> elements. When declaring report fields, the data source should supply data corresponding to all the fields defined in the

report template.

Field Declaration

Field declaration is done as below:

```
<field name="FieldName" />
```

The name attribute

The *name* attribute of the <field> element is mandatory. It references the field in report expressions by name.

The class attribute

The *class* attribute specifies the class name for the field values. Its default value is *java.lang.String*. This can be changed to any class available at runtime. Irrespective of the type of a report field, the engine takes care of casting in the report expressions in which the $F{} token is used, hence making manual casts unnecessary.

Field Description

The <fieldDesciption> element is an optional element. This is very useful when implementing a custom data source, for example. We can store a key or some information, using which we can retrieve the value of field from the custom data source at runtime. By using the <fieldDesciption> element instead of the field name, you can easily overcome restrictions of field-naming conventions when retrieving the field values from the data source.

Following is a piece of JRXML file . Here we can see usage of ***name***, ***class*** and ***fieldDescription*** elements.

```
<field name="country" >
    <fieldDescription><![CDATA[country]]></fieldDescription>
</field>
<field name="name" >
    <fieldDescription><![CDATA[name]]></fieldDescription>
</field>
```

Sort Fields

At the times when data sorting is required and the data source implementation doesn't support it (for e.g. CSV datasource), JasperReports supports in-memory field-based data source sorting. The sorting can be done using one or more <sortField> elements in the report template.

REPORT EXPRESSION

Report expressions are a powerful feature of JasperReports, which allows us to display calculated data on a report. Calculated data is the data that is not a static data and is not specifically passed as a report parameter or datasource field. Report expressions are built from combining report parameters, fields, and static data. By default, the Java language is used for writing report expressions. Other scripting languages for report expressions like Groovy scripting language, JavaScript or BeanShell script are supported by JasperReports compilers.

Expression Declaration

Basically, all report expressions are Java expressions that can reference report fields, report variables and report parameters.

Field Reference In Expression

To use a report field reference in an expression, the name of the field must be put between **$F{** and **}** character sequences, as shown below.

```
<textfieldexpression>
    $F{Name}
</textfieldexpression>
```

Following is a piece of JRXML file:

```
<textFieldExpression >
    <![CDATA[$F{country}]]>
</textFieldExpression>
```

Variable Reference In Expression

To reference a variable in an expression, we must put the name of the variable between **$V{** and **}** like in the example below:

```
<textfieldexpression>
    "Total height : " + $V{SumOfHeight} + " ft."
</textfieldexpression>
```

Parameter Reference In Expression

To reference a parameter in an expression, the name of the parameter should be put between **$P{** and **}** like in the following example:

```
<textfieldexpression>
    "ReportTitle : " + $P{Title}
</textfieldexpression>
```

Following example shows how to extract and display the first letter from java.lang.String report field "Name":

```
<textFieldExpression>
    $F{Name}.substring(0, 1)
</textFieldExpression>
```

Resource Bundle Reference In Expression

To reference a resource in an expression, the *key* should be put between **$R{** and **}** like in the following example:

```
<textfieldexpression>
    $R{report.title}
</textfieldexpression>
```

Based on the runtime-supplied locale and the *report.title* key, the resource bundle associated with the report template is loaded. Hence the title of report is displayed by extracting the String value from the resource bundle.

Calculator

Calculator is an entity in JasperReports, which evaluates expressions and increments variables or datasets at report-filling time. using an instance of net.sf.jasperreports.engine.fill.JRCalculator class.

Java source file is generated and compiled by Java-based report compilers on the fly. This generated class is a subclass of the JRCalculator, and the bytecode produced by compiling it is stored inside the JasperReport object. This bytecode is loaded at the report filling time and the resulting class is instantiated to obtain the calculator object needed for expression evaluation.

Conditional Expressions

Jasper Reports doesn't support if-else statements when defining variable expressions. Instead you can use the ternary operators **{cond} ? {statement 1} : {statement 2}**. You can nest this operator inside a Java expression to obtain the desired output based on multiple conditions.

REPORT VARIABLES

Report variables are special objects built on top of a report expression. Report variables simplify the following tasks:

- Report expressions which are heavily used throughout the report template. These expressions can be declared only once by using the report variables.

- Report variables can perform various calculations based on the corresponding expressions values like: count, sum, average, lowest, highest, variance, etc

If variables are defined in a report design, then these can be referenced by new variables in the expressions. Hence the order in which the variables are declared in a report design is important.

Variable Declaration

A variable declaration is as follows:

```
<variable name="CityNumber"
   incrementGroup="CityGroup" calculation="Count">
   <variableExpression>
       <![CDATA[Boolean.TRUE]]>
   </variableExpression>
</variable>
```

As seen above, <variable> element contains number of attributes. These attributes are summarized below:

The Name Attribute

Similar to *parameters* and *fields*, the *name* attribute of </variable> element is mandatory. It allows referencing the variable by its declared name in report expressions.

The Class Attribute

The *class* attribute is also mandatory and it specifies the class name for the variable values. Its default value is *java.lang.String*. This can be changed to any class available in the classpath, both at report-compilation time and report filling time. Irrespective of the type of a report value, the engine takes care of casting in the report expressions in which the $V{} token is used, hence making manual casts unnecessary.

Calculation

This attribute determines what calculation to perform on the variable when filling the report. The following subsections describe all the possible values for the calculation attribute of the <variable> element.

- *Average:* The variable value is the average of every non-null value of the variable expression. Valid for numeric variables only.

- *Count:* The variable value is the count of non-null instances of the variable expression.

- *First:* The variable value is the value of the first instance of the variable expression. Subsequent values are

ignored.

- *Highest:* The variable value is the highest value for the variable expression.

- *Lowest:* The variable value is the lowest value for the variable expression in the report.

- *Nothing:* No calculations are performed on the variable.

- *StandardDeviation:* The variable value is the standard deviation of all non-null values matching the report expression. Valid for numeric variables only.

- *Sum:* The variable value is the sum of all non-null values returned by the report expression.

- *System:* The variable value is a custom calculation.(calculating the value for that variable yourself, using the scriptlets functionality of JasperReports)

- *Variance:* The variable value is the variance of all non-null values returned by evaluation of a report variable's expression.

Incrementer FactoryClass

This attribute determines the class used to calculate the value of the variable when filling the current record on the report. Default value would be any class implementing **net.sf.jasperreports.engine.fill.JRIncrementerFactory**. The factory class will be used by the engine to instantiate incrementer objects at runtime depending on the *calculation* attribute set for the variable.

IncrementType

This determines when to recalculate the value of the variable. This attribute uses values, as below:

- *Column:* The variable value is recalculated at the end of each column

- *Group:* The variable value is recalculated when the group specified by incrementGroup changes.

- *None:* The variable value is recalculated with every record.

- *Page:* The variable value is recalculated at the end of every page.

- *Report:* The variable value is recalculated once, at the end of the report.

IncrementGroup

This determines the name of the group at which the variable value is recalculated, when *incrementType* is *Group*. This takes name of any group declared in the JRXML report template.

ResetType

This determines when the value of a variable is reset. This attribute uses values, as below:

- *Column:* The variable value is reset at the beginning of each column.

- *Group:* The variable value is reset when the group specified by incrementGroup changes.

- *None:* The variable value is never reset.

- *Page:* The variable value is reset at the beginning of every page.

- *Report:* The variable value is reset only once, at the beginning of the report.

ResetGroup

This determines the name of the group at which the variable value is reset, when *resetType* is *Group*. The values for this attribute would be the name of any group declared in the JRXML report template.

Built-In Report Variables

There are some built-in system variables, ready to use in expressions, as follows:

Variable Name	Description
PAGE_NUMBER	This variable's value is its current page number. It can be used to display both the current page number and the total number of pages using a special feature of JasperReports text field elements, the *evaluationTime* attribute.
COLUMN_NUMBER	This variable contains the current column number
REPORT_COUNT	This report variable contains the total number of records processed.
PAGE_COUNT	This variable contains the number of records that were processed when generating the current page.
COLUMN_COUNT	This variable contains the number of records that were processed when generating the current column.
GroupName_COUNT	The name of this variable is derived from the name of the group it corresponds to, suffixed with the _COUNT sequence. This variable contains the number of records in the current group.

REPORT SECTIONS

Sections are portions of the report that have a specified height and can contain report objects like lines, rectangles, images or text fields.

The report engine iterates through the virtual records of the supplied report data source, at report filling time. Depending on each section's defined behavior , the engine then renders each report section when appropriate. For instance, the detail section is rendered for each record in the data source. When page breaks occur, the page header and page footer sections are rendered as needed.

In JasperReports terminology, report sections are also called report bands. Sections are made of one or more bands. These sections are filled repeatedly at report-generating time and prepare the final document.

Main Sections

A report template in JasperReports has the following main sections:

```
<title></title>

<pageheader></pageheader>

<columnheader></columnheader>

<groupheader></groupheader>
```

```
<detail></detail>

<groupfooter></groupfooter>

<columnfooter></columnfooter>

<pagefooter></pagefooter>

<lastpagefooter></lastpagefooter>

<summary></summary>

<nodata></nodata<>

<background></background>
```

Section, Elements and Attribute Relation

The following diagram shows the elements and attributes relationship in a section in a report.
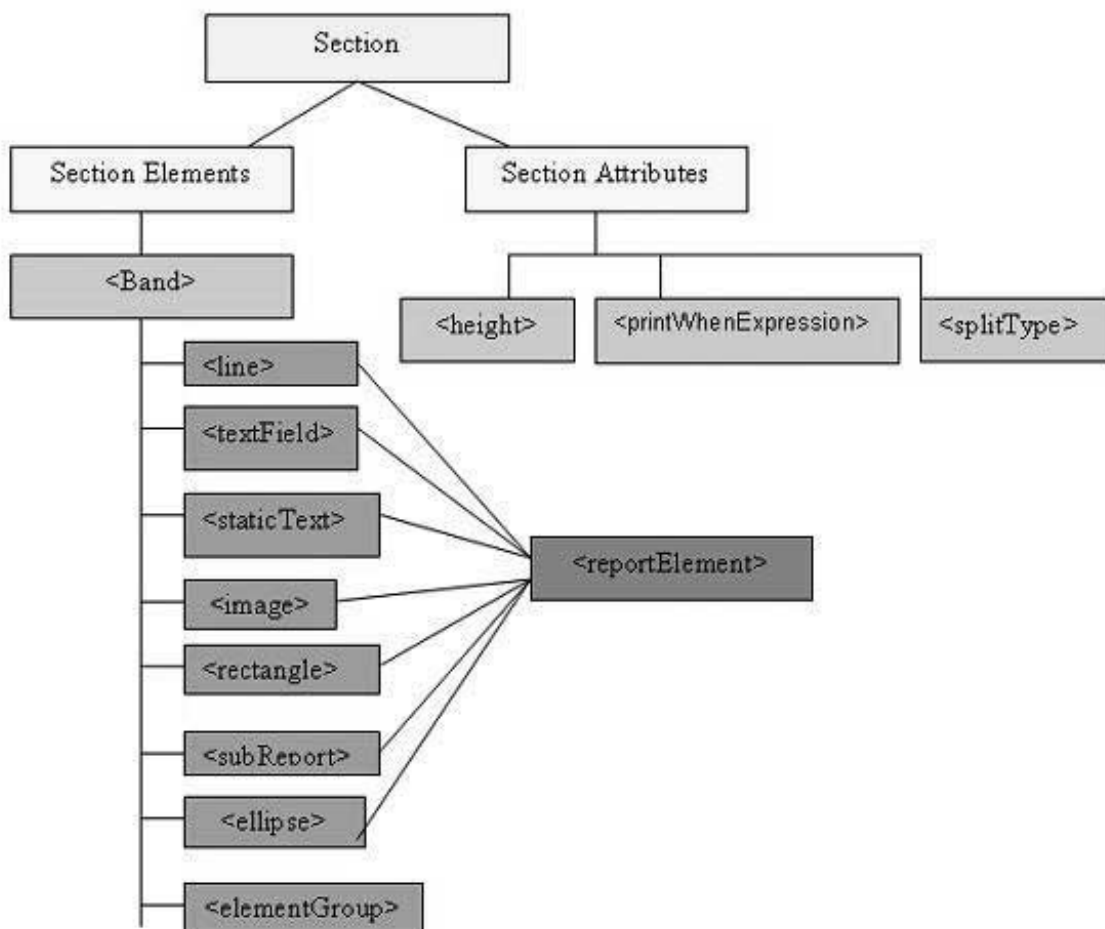
REPORT GROUPS

Groups in JasperReports help organize data on report in a logical manner. A report group represent a sequence of consecutive records in the data source that have something in common, like the value of a certain report field. A report group is defined by the <group> element. A report can have any number of groups. Once declared, groups can be referred to throughout the report.

A report group has three elements:

- *Group expression*: This indicates the data that must change to start a new data group.

- *Group header section*: Helps place label at the beginning of grouped data.

- *Group footer section*: : Helps place label at the end of grouped data.

During the iteration through the data source at report-filling time if the value of the group expression changes , a group rupture occurs and the corresponding <groupFooter> and <groupHeader> sections are inserted in the resulting document.

> *Report group mechanism does not perform any sorting on the data supplied by the data source. Data grouping works as expected only when the records in the data source are already ordered according to the group expressions used in the report.*

Group Attributes

The <group> element contains attributes that allow us to control how grouped data is laid out. The attributes are summarized in table below:

Attribute	Description
name	This is mandatory. It references the group in report expressions by name. It follows the same naming conventions we that we mentioned for the report parameters, fields, and report variables. It can be used in other JRXML attributes when you want to refer a particular report group.
isStartNewColumn	When set to *true*, each data group will begin on a new column. Default value is *false*
isStartNewPage	When set to '*true*, each data group will begin on a new page. Default value is *false*
isResetPageNumber	When set to *true*, the report page number will be reset every time a new group starts. Default value is *false*
isReprintHeaderOnEachPage	When set to *true*, the group header will be reprinted on every page. Default value is *false*
minHeightToStartNewPage	Defines minimum amount of vertical space needed at the bottom of the column in order to place the group header on the current column. The amount is specified in report units.
footerPosition	Renders position of the group footer on the page, as well as its behavior in relation to the report sections that follow it.Its value can be: *Normal*, *StackAtBottom*, *ForceAtBottom*, *CollateAtBottom*. Default value is *Normal*
keepTogether	When set to *true*, prevents the group from splitting on its first break attempt

REPORT FONTS

A report contains text elements and each of these can have its own font settings. These settings can be specified using the **<font>** tag available in the <textElement> tag. A report can define a number of fonts. Once defined, they can be used as default or base font settings for other font definitions throughout the entire report.

Report Fonts

A report font is a collection of font settings, declared at the report level. A report font can be reused throughout the entire report template when setting the font properties of text elements.

> *Report fonts are now deprecated. Do not use <reportFont/> elements declared within the document itself. Use the <style/> element instead.*

Font Attributes

Table below summarizes the main attributes of the **<font>** element:

Attribute	Description
fontName	The font name, which can be the name of a physical font, a logical one or the name of a font family from the registered JasperReports font extensions.
size	The size of the font measured in points. It defaults to 10.
isBold	The flag specifying if a bold font is required. It defaults to false.
isItalic	The flag specifying if an italic font is required. It defaults to false.
isUnderline	The flag specifying if the underline text decoration is required. It defaults to false.
isStrikeThrough	The flag specifying if the strikethrough text decoration is required. It defaults to false.
pdfFontName	The name of an equivalent PDF font required by the iText library when exporting documents to PDF format.
pdfEncoding	The equivalent PDF character encoding, also required by the iText library.
isPdfEmbedded	The flag that specifies whether the font should be embedded into the document itself. It defaults to false. If set to true, helps view the PDF document without any problem.

Font Types

In JasperReports fonts can be categorized in the following types:

1. **Logical Fonts:** These fonts are the five font types that have been recognized by the Java platform since version 1.0: **Serif, SansSerif, Monospaced, Dialog, and DialogInput**. These logical fonts are not actual font libraries that are installed anywhere on the system. They are merely font type names recognized by the Java runtime. These must be mapped to some physical font that is installed on the system.

2. **Physical Fonts:** These fonts are the actual font libraries consisting of, for example, TrueType or PostScript Type 1 fonts. The physical fonts may be Arial, Time, Helvetica, Courier, or any number of other fonts, including international fonts.

3. **Font Extensions:** The JasperReports library can make use of fonts registered on-the-fly at runtime, through its built-in support for font extensions. A list of font families can be made available to the JasperReports using font extension. These are made out of similarly looking font faces and supporting specific locales.

As described in the table above we need to specify in the attribute *fontName* the name of a physical font, the name of a logical font, or the name of a font family from the registered JasperReports font extensions.

PDF Font Name

JasperReports library uses the iText library, when exporting reports to PDF(Portable Document Format). PDF files can be viewed on various platforms and will always look the same. This is partially because in this format there is a special way of dealing with fonts. *fontName* attribute is of no use when exporting to PDF. Attribute *pdfFontName* exist where we need to specify the font settings.

Default Fonts and Inheritance

Each text element inherits font and style attributes from its parent element which in turn inherits these attributes from its parent. If no styles and/or fonts are defined for elements, the default style (and/or font - but this is now deprecated) declared in the <jasperReport/> root element will be applied.

Defining default styles or fonts in JasperReports is not mandatory. If no font is defined for a given element, the engine looks either for the inherited font attributes, or, if no attributes are found on this way, it looks for the *net.sf.jasperreports.default.font.name* property in the */src/default.jasperreports.properties* file. Its value defines the name of the font family to be used when font properties are not explicitly defined for a text element or inherited from its parent.

The main default font properties and their values defined in the */src/default.jasperreports.properties* file are in the table below:

Property	Description
net.sf.jasperreports.default.font.name=SansSerif	The default font name.
net.sf.jasperreports.default.font.size=10	The default font size.
net.sf.jasperreports.default.pdf.font.name=Helvetica	The default PDF font.
net.sf.jasperreports.default.pdf.encoding=Cp1252	The default PDF character encoding.
net.sf.jasperreports.default.pdf.embedded=false	By default PDF fonts are not embedded.

UNICODE SUPPORT

One of the main features concerning the text content in a given report is the possibility to internationalize it. It means we can run the report in different localized environments, using different languages and other localization settings without any hardcoded modification. Character encoding is an important feature when a report is intended to be internationalized.

Character Encoding

A character is the smallest unit of writing that's capable of conveying information. It's an abstract concept, a character does not have a visual appearance. "Uppercase Latin A" is a different character from "lowercase Latin a" and from "uppercase Cyrillic A" and "uppercase Greek Alpha".

A visual representation of a character is known as a *glyph*. A certain set of glyphs is called a *font*. "Uppercase Latin A", "uppercase Cyrillic A" and "uppercase Greek Alpha" may have identical glyphs, but they are different characters. At the same time, the glyphs for "uppercase Latin A" can look very different in Times New Roman, Gill Sans and Poetica

chancery italic, but they still represent the same character.

The set of available characters is called a *character repertoire*. The location (index) of a given character within a repertoire is known as its code position, or code point. The method of numerically representing a code point within a given repertoire is called the **character encoding**.

Encodings are normally expressed in terms of octets. An octet is a group of eight binary digits, i.e., eight ones and zeros. An octet can express a numeric range between 0 and 255, or between 0x00 and 0xFF, to use hexadecimal notation.

Unicode

A Unicode is a character repertoire that contains most of the characters used in the languages of the world. It can accommodate millions of characters, and already contains hundreds of thousands. Unicode is divided into "planes" of 64K characters. The only one used in most circumstances is the first plane, known as the basic multilingual plane, or BMP.

UTF-8 is the recommended encoding. It uses a variable number of octets to represent different characters.

In a JRXML file the encoding attribute is specified in the header. It is used at report compilation time to decode the XML content. For instance, if the report contains French words only and characters such as ç, é, â , then the ISO-8859-1 (a.k.a Latin-1) encoding is sufficient:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

As seen above, ideally we can choose the encoding fit to the minimal character set which can correctly represent all the characters in the document. But in case of multilanguage documents (i.e documents containing words spelled in several languages), one should choose the encoding adapted to the minimal character set able to correctly represent all the characters in the document, even if they belong to different languages. One of the character encodings able to handle multilingual documents is the **UTF-8**, used as default encoding value by JasperReports.

The texts are usually kept in resource bundle files rather than within the document during internationalization. So, there are cases where the JRXML itself looks completely ASCII-compatible, but generated reports at runtime do contain texts unreadable with ASCII. As a result, for a certain type of document export formats (such as CSV, HTML, XHTML, XML, text) one has to know the encoding for the generated document too. Different languages are supported by different character encodings, so each time we need to run a report in a localized environment, we have to know which is the most appropriate character encoding for the generated document language. In this case, the encoding property defined in the JRXML file itself might be no more useful.

To solve this kind of issues an export custom property : *net.sf.jasperreports.export.character.encoding* which defaults to UTF-8 is present in Jasperreport. This default value is set in the *default.jasperreports.properties* file. For more specific options at export time, the CHARACTER_ENCODING export parameter is also available.

REPORT STYLES

JasperReports has a feature <style> which helps controls text properties in a report template. This element is a collection of style settings declared at the report level. Properties like foreground color, background color, whether the font is bold, italic, or normal, the font size, a border for the font, and many other attributes are controlled by <style> element. Styles can extend other styles, and add to, or override properties of the parent style.

Style Properties

A <style> element has many attributes. Some of the most commonly used are listed in the table below:

Attribute	Description
name	Is mandatory. It must be unique because it references the corresponding report style throughout the report
isDefault	Indicates whether this style is the document's default style.
style	A reference to the parent style
mode	Specifies the element's transparency. Possible values are *Opaque* and *Transparent*.
forecolor	The foreground color of object.
backcolor	The background color of object.
fill	Determines the fill pattern used to fill the object. At the moment the single value allowed is *Solid*.
radius	Specifies the radius of the rectangle's corner arc.
scaleImage	Specifies the scale for images only. Possible values: *Clip, FillFrame, RetainShape, RealHeight, RealSize*.
hAlign	Specifies the horizontal alignment. Possible values: *Left, Center, Right, Justified*.
vAlign	Specifies the vertical alignment. Possible values: *Top, Middle, Bottom*.
rotation	Specifies the element's rotation. Possible values: *None, Left, Right, UpsideDown*.
lineSpacing	Specifies the line spacing between lines of text. Possible values: *Single, 1_1_2, Double*.
markup	Specifies the markup style for styled texts
fontName	Specifies the font name.
fontSize	Specifies the font size.
isBold	Indicates if the font style is bold
isItalic	Indicates if the font style is italic.
isUnderline	Indicates if the font style is underline
isStrikeThrough	Indicates if the font style is strikethrough.
pdfFontName	Specifies the related PDF font name.
pdfEncoding	Specifies the character encoding for the PDF output format
isPdfEmbedded	Indicates if the PDF font is embedded.
pattern	Specifies the format pattern for formatted texts.
isBlankWhenNull	Indicates if a white space should be shown if the text is not present.

Conditional Styles

In some situations, a style should be applied only when certain condition is met (for example, to alternate adjacent row colors in a report detail section). This can be achieved using conditional styles.

A conditional style has two elements:

- a Boolean condition expression
- a style

Style Templates

We can make a set of reports with a common look by defining the style at a common place. This common style template can then be referenced by the report templates. A style template is an XML file that contains one or more style definitions. Style template files use by convention the ***.jrtx** extension, but this is not mandatory. A style template contains following elements:

- *<jasperTemplate>*:This is the root element of a style template file.

- *<template>*: This element is used to include references to other template files. The contents of this element are interpreted as the location of the referred template file.

- *<style>*: This element is identical to the element with the same name from report design templates (JRXML files), with the exception that a style in a style template cannot contain conditional styles. This limitation is caused by the fact that conditional styles involve report expressions, and expressions can only be interpreted in the context of a single report definition.

REPORT SCRIPTLETS

Scriptlets are sequences of Java code that are executed every time a report event occurs. Values of report variables can be affected through scriptlets.

Scriptlet Declaration

We can declare a scriptlet in two ways:

- Using **<scriptlet>** element. This element has *name* attribute and *class* attribute. The *class* attribute should specify the name of the class, which extends *JRAbstractScriptlet* class. The class must be available in the classpath at report filling time and must have an empty constructor, so that the engine can instantiate it on the fly.

- Using the attribute **scriptletClass** of the element **<jasperReport>**, in the report template (JRXML). By setting this attribute with fully qualified name of scriptlet (including the entire package name), we indicate that we want to use a scriptlet. The scriptlet instance created with this attribute, acts like the first scriptlet in the list of scriptlets and has the predefined name REPORT.

Scriptlet class

A scriptlet is a java class which must extend either of the following classes:

- **net.sf.jasperreports.engine.JRAbstractScriptlet**: This class contains a number of abstract methods that must be overridden in every implementation. These methods are called automatically by JasperReports at the appropriate moment. Developer must implement all the abstract methods.

- **net.sf.jasperreports.engine.JRDefaultScriptlet**: This class contains default empty implementations of every method in JRAbstractScriptlet. A developer is only required to implement those methods he/she needs for their project.

Following table lists the methods in the above class. These methods will be called by the report engine at the appropriate time, during report filling phase.

Method	Description
public void beforeReportInit()	Called before report initialization.
public void afterReportInit()	Called after report initialization.
public void beforePageInit()	Called before each page is initialized.
public void afterPageInit()	Called after each page is initialized.
public void beforeColumnInit()	Called before each column is initialized.
public void afterColumnInit()	Called after each column is initialized.
public void beforeGroupInit(String groupName)	Called before the group specified in the parameter is initialized.
public void afterGroupInit(String groupName)	Called after the group specified in the parameter is initialized.
public void beforeDetailEval()	Called before each record in the detail section of the report is evaluated.
public void afterDetailEval()	Called after each record in the detail section of the report is evaluated.

Any number of scriptlets can be specified per report. If no scriptlet is specified for a report, the engine still creates a single JRDefaultScriptlet instance and registers it with the built-in REPORT_SCRIPTLET parameter.

We can add any additional methods we need to our scriptlets. Reports can call these methods by using the built-in parameter REPORT_SCRIPTLET.

Global Scriptlets

We can associate scriptlets in another way to reports, which is by declaring the scriptlets globally. This makes the scriptlets apply to all reports being filled in the given JasperReports deployment. This is made easy by the fact that scriptlets can be added to JasperReports as extensions. The scriptlet extension point is represented by the *net.sf.jasperreports.engine.scriptlets.ScriptletFactory* interface. JasperReports will load all scriptlet factories available through extensions at runtime. Then, it will ask each one of them for the list of scriptlets instances that they want to apply to the current report that is being run. When asking for the list of scriptlet instances, the engine gives some context information that the factory could use in order to decide which scriptlets actually apply to the current report.

Report Governors

Governors are just an extension of global scriptlets that enable us to tackle a problem of report engine entering infinite loop at runtime, while generating reports. Invalid report templates cannot be detected at design time, because most of the time the conditions for entering the infinite loops depend on the actual data that is fed into the engine at runtime. Report Governors help deciding whether a certain report has entered an infinite loop and they can stop it, preventing resource exhaustion for the machine that runs the report.

JasperReports has two simple report governors that would stop a report execution based on a specified maximum

number of pages or a specified timeout interval. They are:

1. **net.sf.jasperreports.governors.MaxPagesGovernor**: This is a global scriptlet that is looking for two configuration properties to decide if it applies or not to the report currently being run. The configuration properties are:

   - net.sf.jasperreports.governor.max.pages.enabled=[true|false]

   - net.sf.jasperreports.governor.max.pages=[integer]

2. **net.sf.jasperreports.governors.TimeoutGovernor**: This is also a global scriptlet that is looking for the following two configuration properties to decide if it applies or not: The configuration properties are:

   - net.sf.jasperreports.governor.timeout.enabled=[true|false]

   - net.sf.jasperreports.governor.timeout=[milliseconds]

The properties for both governors can be set globally, in the jasperreports.properties file, or at report level, as custom report properties. This is useful because different reports can have different estimated size or timeout limits and also because you might want turn on the governors for all reports, while turning it off for some, or vice-versa.

CREATE SUBREPORTS

Subreports are like normal report templates. They are in fact *net.sf.jasperreports.engine.JasperReport* objects, which are obtained after compiling a *net.sf.jasperreports.engine.design.JasperDesign object*.

<subreport> Element

A <subreport> element is used when introducing subreports into master reports. Here is the list of sub-elements in the <subreport> JRXML element.

- <reportElement>

- <parametersMapExpression> : This is used to pass a map containing report parameters to the subreport. The map is usually obtained from a parameter in the master report, or by using the built-in REPORTS_PARAMETERS_MAP parameter to pass the parent report's parameters to the subreport. This expression should always return a *java.util.Map* object in which the keys are the parameter names.

- <subreportParameter> : This element is used to pass parameters to the subreport. It has an attribute *name*, which is mandatory.

- <connectionExpression > : This is used to pass a *java.sql.Connection* to the subreport. It is used only when the subreport template needs a database connection during report filling phase.

- <dataSourceExpression> : This is used to pass a datasource to the subreport. This datasource is usually obtained from a parameter in the master report or by using the built-in REPORT_DATA_SOURCE parameter to pass the parent report's datasource to the subreport.

  The elements (*connectionExpression and dataSourceExpression*) cannot be present at the same time in a <subreport> element declaration. This is because we cannot supply both a data source and a connection to the subreport. We must decide on one of them and stick to it.

- <returnValue> : This is used to assign the value of one of the subreport's variables to one of the master report's variables. This sub element has attributes as follows:

  - *subreportVariable*: This attribute specifies the name of the subreport variable whose value is to be returned.

- *toVariable*: This attribute specifies the name of the parent report variable whose value is to be copied/incremented with the value from the subreport.

- *calculation*: This attribute can take values : Nothing, Count, DistinctCount, Sum, Average, Lowest, Highest, StandardDeviation, Variance. Default value for attribute *calculation* is "Nothing".

- *incrementerFactoryClass*: This attribute specifies the factory class for creating the incrementer instance.

- <subreportExpression> : This indicates where to find the compiled report template for the subreport. This element has a **class** attribute. The *class* attribute can take any of these values:java.lang.String, java.io.File, java.net.URL, java.io.InputStream, net.sf.jasperreports.engine.JasperReport.Default value is *java.lang.String*.

- isUsingCache : This is an attribute of the <subreport> element. This is a Boolean, when set to *true*, the reporting engine will try to recognize previously loaded subreport template objects, using their specified source. This caching functionality is available only for subreport elements that have expressions returning java.lang.String objects as the subreport template source, representing file names, URLs, or classpath resources.

CREATING CHARTS

Using the new chart component, user need to apply only the visual settings and define expressions that will help build the chart dataset. JasperReports uses JFreeChart as the underlying charting library. When configuring a new chart component, following three components are involved:

- The overall chart component.

- The chart dataset (which groups chart data-related settings).

- The chart plot (which groups visual settings related to the way the chart items are rendered).

JasperReports currently supports the following types of charts: Pie, Pie 3D, Bar, Bar 3D, XY Bar, Stacked Bar, Stacked Bar 3D, Line, XY Line, Area, XY Area, Stacked Area, Scatter, Bubble, Time Series, High-Low-Open-Close, Candlestick, Multiple Axis, Meter, Thermometer and Gantt.

Chart Properties

Charts are normal report elements, so they share some of their properties with all the other report elements. There is a JRXML element called **<chart>**, used to create each type of chart. This element groups special chart-specific settings that apply to all types of charts.

Chart Sub-Elements

The sub-elements of <chart> element are:

- **<reportElement>**: These are displayable objects like static texts, text fields, images, lines, and rectangles that you put in your report template sections

- **<Box>**: This element is used to surround charts by a border that's customizable on each side.

- **<chartTitle>**: This element is used to place the title of the chart. The *position* attribute decides the title position of the chart in the report. This element has attributes - **Position** (Values could be *Top*, *Bottom*, *Left*, *Right*. Deafult value is *Top*), **color**. <chartTitle> has *font* and *titleExpression* as subelements.

- **<chartSubtitle>**: This element is used to place the subtitle of the chart. This element has attribute - **color**. <chartSubtitle> has *font* and *subtitleExpression* as subelements.

- **<chartLegend>**: The element can control the font-related properties as well as the text color and the background color of the chart legend using this element. This element has attributes - **textColor**, **backgroundColor**

- **<anchorNameExpression>**: This element creates the target for the anchor.

- **<hyperlinkReferenceExpression>**: This element contains a report expression indicating the name of the external resource (usually a URL).

- **<hyperlinkAnchorExpression>**: Hyperlink points to an anchor in an external resource.

- **<hyperlinkPageExpression>**: Hyperlink points to a page in the current report.

- **<hyperlinkTooltipExpression>**: This element controls the ToolTip of hyperlink. The type of the expression should be *java.lang.String*.

- **<hyperlinkParameter>**: This element when present generates a final hyperlink depending on the parameter values.

Chart attributes

Attributes in the <chart> element available for all chart types are:

- **isShowLegend**: This attribute is used to determine if a chart legend will be displayed on the report. Values could be *true*, *false*. Default value is *true*

- **evaluationTime**: Determines when the chart's expression will be evaluated. Values could be *Now*, *Report*, *Page*, *Column*, *Group*, *Band*. Default value is *Now*.

- **evaluationGroup**: This attribute determines the name of the group to be used to evaluate the chart's expressions. The value for this attribute must match the name of the group we would like to use as the chart's evaluation group.

- **hyperlinkType**: This attribute can hold any text value. Default value is *None*. This means neither the text fields nor the images represent hyperlinks, even if the special hyperlink expressions are present.

- **hyperlinkTarget**: This attribute help customize the behavior of the specified link when it is clicked in the viewer. Values could be *Self*, *Blank*. Default value is *Self*

- **bookmarkLevel**: This attribute when set to a positive integer, generate bookmarks in reports exported to PDF. Deafult value is *0*.

- **customizerClass**: This is the name of a class (optional) that can be used to customize the chart. The value for this element must be a String containing the name of a customizer class.

Chart Datasets

One of the common properties across all chart types is **<dataset>** element. Chart datasets help mapping report data and retrieving chart data at runtime. Each chart type contains different sub-elements to define a chart's expressions that define the data used to generate the chart. All of these sub-elements contain a <dataset> element that defines when the chart's expressions are evaluated and reset.

Several types of chart datasets are available in JasperReports because each type of chart works with certain datasets: Pie, Category, XY, Time Series, Time Period, XYZ, and High-Low. Each of these dataset types implements *net.sf.jasperreports.engine.JRChartDataset* interface that define chart datasets. All chart datasets initialize and increment in the same way, and differ only in the type of data or data series they map.

Dataset Properties

Table below summarizes the attributes of the element <dataset>

Attribute	Description	Values
resetType	This attribute determines when the value of the chart expression is to be reset.	None, Report, Page, Column, Group. Default value is **Report**
resetGroup	this attribute determines the name of the group at which the chart expression value is reset.	The value for this attribute must match the name of any group declared in the JRXML report template.
incrementType	This attribute determines when to recalculate the value of the chart expression.	None, Report, Page, Column, Group. Default value is **"None"**.
incrementGroup	This attribute determines the name of the group at which the chart expression is recalculated.	The value for this attribute must match the name of a group declared in the JRXML report template.

Table below summarizes the sub elements of the element <dataset>

Sub element	Description
<incrementWhenExpression>	The way a chart dataset is incremented can be customized by filtering out unwanted data through the use of this sub element.
<datasetRun>	This contains information required to instantiate a report subdataset.

Chart Plots

Another common JRXML element through all chart types is the **<plot>** element. This allows us to define several of chart's characteristics like orientation and background color. Plots differ based on the type of chart.

Plot Attribute

The table below summarizes the attributes of <plot> element.

Attribute	Description	Values
backcolor	This attribute defines the chart's background color.	Any six digit hexadecimal value is a valid value for this attribute. The hexadecimal value must be preceded by a #.
orientation	This attribute defines the chart's orientation.	Horizontal,Vertical Default value is **"Vertical"**
backgroundAlpha	This attribute defines the transparency of the chart's background color.	The valid values for this attribute include any decimal number between 0 and 1, inclusive. The higher the number, the less

		transparent the background will be. Default value is **"1"**.
foregroundAlpha	This attribute defines the transparency of the chart's foreground colors.	The valid values for this attribute include any decimal number between 0 and 1, inclusive. The higher the number, the less transparent the background will be. Default value is **"1"**.
labelRotation	This attribute allows rotation of text labels on x-axis to rotate clockwise or anti-clockwise. This attribute applies only to charts for which the x axis is not numeric or does not display dates.	Default value is **"0.0"**.

The <plot> element has a subelement <seriesColor> whose attributes are: *seriesOrder* and *color*. This element customizes colors for series, and their position within in the color sequence.

CROSSTABS

Crosstab (cross-tabulation) reports are reports containing tables that arrange data across rows and columns in a tabular form. Crosstab object is used for inserting a crosstab report within the main report. Crosstabs can be used with any level of data (nominal, ordinal, interval, or ratio), and usually display the summarized data, contained in report variables, in the form of a dynamic table. Variables are used to display aggregate data such as sums, counts, average values.

Crosstab Properties

JRXML element **<crosstab>** is used to insert a crosstab into a report.

Attribute

Following is a list of attribute of a **<crosstab>** element:

- **isRepeatColumnHeaders**: Indicates whether the column headers should be reprinted after a page break. The default value is *true*.

- **isRepeatRowHeaders**: Indicates whether the row headers should be reprinted after a crosstab column break. The default value is *true*.

- **columnBreakOffset**: When a column break occurs, indicates the amount of vertical space, measured in pixels, before the subsequent crosstab piece to be placed below the previous one on the same page. The default value is 10.

- **runDirection**: Indicates whether the crosstab data should be filled from left to right (LTR) or from right to left (RTL). The default value is LTR.

- **ignoreWidth**: Indicates whether the crosstab will stretch beyond the initial crosstab width limit and don't generate column breaks. Else it will stop rendering columns within the crosstab width limit and continue with the remaining columns only after all rows have started rendering. The default value is *false*.

Sub Elements

A <crosstab> element has following sub elements :

<reportElement>, <crosstabParameter>, <parametersMapExpression>, <crosstabDataset>, <crosstabHeaderCell>, <rowGroup>, <columnGroup>, <measure>, <crosstabCell>, and <whenNoDataCell>

Data Grouping in Crosstab

The crosstab calculation engine aggregates data by iterating through the associated dataset records. In order to aggregate data, one need to group them first. In a crosstab, rows and columns are based on specific group items, called **buckets**. A bucket definition should contain:

- *bucketExpression*: The expression to be evaluated in order to obtain data group items.

- *comparatorExpression*: Needed in the case the natural ordering of the values is not the best choice.

- *orderByExpression*: Indicates the value used to sort data.

Row and column groups (defined above) in a crosstab rely on **buckets**.

Built-In Crosstab Total Variables

Below is a list of current value of measure and totals of different levels corresponding to the cell can be accessed through variables named according to the following scheme:

- The current value of a measure calculation is stored in a variable having the same name as the measure.

- *<Measure>_<Column Group>_ALL*: This yields the total for all the entries in the column group from the same row..

- *<Measure>_<Row Group>_ALL*: This yields the total for all the entries in the row group from the same column.

- *<Measure>_<Row Group>_<Column Group>_ALL*: This yields the combined total corresponding to all the entries in both row and column groups.

INTERNATIONALIZATION

At times we need reports in different languages. Writing the same report for each different language implies a lot of redundant work. Only pieces of text differing from language to language should be written separately, and loaded into text elements at runtime, depending on locale settings. This is the purpose of the report internationalization. Internationalized reports once written can run everywhere.

In the following steps, we have listed how to generate a report in different languages and also some other features of report internationalization:

- Associate a resource bundle *java.util.ResourceBundle* with the report template. There are two ways to associate the java.util.ResourceBundle object with the report template.

  - At design time, by setting the *resourceBundle* attribute of the report template object to the base name of the target resource bundle.

  - A dynamic/runtime association can be made by supplying a *java.util.ResourceBundle* object as the value for the REPORT_RESOURCE_BUNDLE parameter at report-filling time.

  - If the report needs to be generated in a locale that is different from the current one, the built-in REPORT_LOCALE parameter can be used to specify the runtime locale when filling the report.

- To facilitate report internationalization, a special syntax **$R{}** is available inside report expressions to reference *java.lang.String* resources placed inside a *java.util.ResourceBundle* object associated with the report. The **$R{}**

character syntax extracts the locale-specific resource from the resource bundle based on the key that must be put between the brackets:

```
<textFieldExpression>
$R{report.title}
</textFieldExpression>
```

The above text field displays the title of the report by extracting the String value from the resource bundle associated with the report template based on the runtimesupplied locale and the *report.title* key

- Formatting messages in different languages based on the report locale, there's a built-in method inside the report's *net.sf.jasperreports.engine.fill.JRCalculator*. This method offers functionality similar to the *java.text.MessageFormat* class. This method, msg(), has three convenient signatures that allow you to use up to three message parameters in the messages.

- A built-in *str( )* method (the equivalent of the $R{} syntax inside the report expressions), which gives access to the resource bundle content based on the report locale.

- For date and time formatting, the built-in REPORT_TIME_ZONE parameter can be used to ensure proper time transformations.

- In the generated output, the library keeps information about the text run direction so that documents generated in languages that have right-to-left writing (like Arabic and Hebrew) can be rendered properly.

- If an application relies on the built-in Swing viewer to display generated reports, then it needs to be internationalized by adapting the button ToolTips or other texts displayed. This is very easy to do since the viewer relies on a predefined resource bundle to extract locale-specific information. The base name for this resource bundle is *net.sf.jasperreports.view.viewer*